Parallel implementation of Artificial Neural Network training for speech recognition

Parallel implementation of Artificial Neural Network training for speech recognition

Stefano Scanzio*, Sandro Cumani*, Roberto Gemello^, Franco Mana^, P. Laface*

* Dipartimento di Automatica e Informatica, Politecnico di Torino,
Corso Duca degli Abruzzi 24, Torino 10129, Italy
^Loquendo S.pA., Via Olivetti 6, Torino 10148, Italy

Abstract

In this paper we describe the implementation of a complete ANN training procedure using the block mode back-propagation learning algorithm for sequential patterns - such as the observation feature vectors of a speech recognition system - exploiting the high performance SIMD architecture of GPU using CUDA and its C-like language interface. We also compare the speed-up obtained implementing the training procedure only taking advantage of the multi-thread capabilities of multi-core processors. In our implementation we take into account all the peculiar aspects of training large scale sequential patterns, in particular, the re-segmentation of the training sentences, the block size for the feed-forward and for the back-propagation steps, and the transfer of huge amount of data from host memory to the GPU card.
Our approach has been tested by training acoustic models for large vocabulary speech recognition tasks, showing a 6 times reduction of the time required to train real-world large size networks with respect to an already optimized implementation using the Intel MKL libraries.
Thanks to these optimizations and to the support of the GPU, the training time for language having a huge set of training sentences (about one million for Italian) can be reduced from approximately a month to 5 days.

Key words: Artificial Neural Network, Block Back-propagation, Focused Attention Back-Propagation, GPU, CUDA, Fast Training

1. Introduction

State of the art speech recognition systems are based on acoustic-phonetic models of the words. Each acoustic unit is modeled by one or more states of a Hidden Markov Model (HMM), a stochastic automaton that characterizes the spectral properties of a sequence of acoustic patterns. Gaussian Mixture Models (GMMs) are often used within each state to model the probability density of the acoustic patterns associated to that state (Rabiner and Juang, 1993).
Acoustic scoring often accounts for most of the processing time in the GMM-HMM approach. In some tasks, the computation of the GMM's emission probabilities can consume 70-80% of the whole decoding process. Many efforts, thus, have been spent to speed-up these computations either by means of Gaussian selection techniques (Chan et

al., 2004) or, recently, by taking advantage of the computational power of Graphics Processing Units (GPU) (Dixon et al., 2009, Cardinal et al., 2009).
An attractive alternative to Gaussian mixture modeling is the use of an Artificial Neural Network (ANN) trained to estimate the posterior probability of each state given an acoustic pattern (Bourlard and Morgan, 1993). The main advantage of using a hybrid ANN-HMM approach in large vocabulary speech recognition is that the computation of the posterior probabilities of the HMM states takes a small fraction of the search time, moreover the ANN models are inherently discriminative.
The most widely used ANNs in speech recognition are feed-forward Multi Layer Perceptron (MPL) networks trained by the error back-propagation paradigm (Rumelhart et al., 1986). This learning procedure relies on iterative gradient descent optimizations to minimize the network errors. The re-estimation of the ANN weights can be performed after all the training patterns have been processed (epoch back-propagation), with a possible high degree of parallelism of the computation. However, the speed of convergence and the accuracy of the model obtained using this procedure are usually inferior compared with the on-line learning procedure where the network weights are updated after each pattern has been processed (Hertz et al., 1991). The latter approach, on the other hand, is intrinsically sequential and not suited for particularly effective parallel computations. A solution that has demonstrated to achieve fast convergence and high accuracy without losing the advantages of parallel computation is the so called block or bunch mode back-propagation, in which the re-estimation of the ANN weights is performed using multiple rather than single training patterns (Anguita et al., 1994).

The core computation in MLPs, both in the feed-forward and in the back-propagation steps, is the inner product of a weight vector and of a feature vector (activation or error vector respectively). Several works have been published that exploit matrix multiplication to convert many inner-product operations into a single matrix multiply operation. Oh and Jung (2004) efficiently perform the matrix multiplications using a GPU to improve the performance of a text detection system for image data and video documents. Jang et al. (2008) perform the same task comparing the performance of a CUDA and OpenMP implementation of the feed-forward step in a three-layer MLP. Lahabar et al. (2008) describe the implementation of epoch mode back-propagation by using CUDA (Compute Unified Device Architecture) and the CUBLAS library (NVIDIA, 2008).
All these works focus on the classification of static patterns.

In this paper we describe the implementation of the block mode back-propagation learning algorithm for sequential patterns - such as the observation feature vectors of a speech recognition system - exploiting the high performance SIMD architecture of GPU using CUDA and its C-like language interface. We also compare the speed-up obtained implementing the training procedure taking advantage of the multi-thread capabilities of multi-core processors only. In our implementation we take into account all the peculiar aspects of training large scale sequential patterns, in particular, the re-segmentation of the training sentences, the block size for the feed-forward and for the back-propagation steps in combination with a focused-attention procedure that allows to dramatically reduce the training times for large datasets, and the transfer of huge amount of data.

The paper is organized as follows. In Section 2, we briefly present the hybrid ANN-HMM models used by the Loquendo ASR speech recognition decoder (Loquendo ASR, 2009). Section 3 recalls the training steps of the ANN-HMM models. Sections 4 and 5 detail the techniques that can be used to accelerate the network training using single or multi-core CPUs, and GPU respectively. The experimental results are summarized in Section 6. Concluding remarks are reported in Section 7.

2. **Hybrid ANN-HMM models**

The Loquendo Automatic Speech Recognizer (Loquendo ASR) is based on hybrid ANN-HMM models, detailed in Fissore et al. (1995) and Albesano et al. (1997), where the acoustic models are left-to-right HMMs, with all the states having the same transition probabilities. The emission probabilities of the HMM states are computed by a three layer MLP. Having two hidden layers, rather than a single larger layer, has the advantage of reducing the total number of connections without any performance degradation. A network is fed with a temporal window including a sequence of the input patterns. A pattern consists of 7 frames, the central frame and three frames for the left and right contexts respectively. A frame is a spectral vector of 39 acoustic features computed every 10 ms. The input layer, thus, has 273 nodes. The first and second hidden layers, have a number of nodes varying from 300 to 500, all with sigmoid activation function. The softmax function is applied to the output layer, which includes a language dependent number of nodes (in the range 700 - 1000) representing the states of a set of context-dependent phonetic units.

3. **ANN-HMM model training**

The training databases usually do not include phoneme utterances, but rather utterances of sentences or words collected from many different speakers, transcribed in terms of their corresponding phonetic units. Training hybrid ANN-HMM models, thus, requires these two alternate steps:

- Find the best alignment of the utterance frames to the states of the corresponding phonetic units.
- Find the weights of the network that better discriminate among the states, by producing for each state an estimate of the posterior probability P(state|**x**), where **x** is the input pattern.

More precisely, the iterative train procedure follows these steps:
Initialization:

- Initialize the network weights with small random values.
- Use a bootstrap alignment of the training sentence patterns to the network output states to obtain the target of the neural network for each input vector. This initial segmentation can be obtained by a raw alignment of the input patterns to the sequence of the states corresponding to the transcription of a sentence, or by a forced

alignment Viterbi recognition procedure (Rabiner and Juang, 1993) using simpler pre-existing models.

Iteration:

- Update the network weights by back-propagation of the errors, using the current alignment
- Update the alignment of the train set patterns by Viterbi forced alignment using the emission probabilities of the updated network.

In the following we will analyze the feed-forward and back-propagation training steps for a network having softmax output nodes and sigmoid activation functions for all the hidden nodes. The cross-entropy error function is used to compute the error between the output of the ANN and the target vector $\mathbf{t}$.

3.1 Feed-forward step

The feed-forward step computes, for each layer, the outputs of the corresponding nodes given the layer input vector $\mathbf{X}_t = (\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_n)_t$ as:

$$net_i = b_i + \sum_{j=1}^{n} w_{ij} \cdot x_j \tag{1}$$

$$out_i = f\left(net_i\right) \tag{2}$$

where n is the number of nodes of the current layer, and $f(net_i)$ is the sigmoid function (3) for all the hidden layers, and the softmax function (4) for the output layer

$$f_{\text{sigmoid}}(net_i) = \frac{1}{1 + \exp(-net_i)} \tag{3}$$

$$f_{\text{softmax}}(net_i) = \frac{\exp(net_i)}{\sum_{j=1}^{n} \exp(net_j)} \tag{4}$$

3.2 Back-propagation

After the evaluation of the outputs produced by the feed-forward step, the net weights are updated on the basis of the error between the target vector and the network outputs by means of the back-propagation algorithm. An error function that is appropriate for dealing with probabilities is the cross-entropy error:

$$E_i = -t_i \cdot \ln\left(out_i\right) - \left(1 - t_i\right) \cdot \ln\left(1 - out_i\right) \tag{5}$$

The back-propagation algorithm proceeds by propagating the error from the output to the underlying hidden layers in order to correct the network weights. The procedure can be summarized in the following three steps:

1. Compute the derivative of the error with respect to the input for every node i

$$\delta_i^{(layer)} = \frac{\partial E_i}{\partial net_i^{(layer)}} = \frac{\partial E_i}{\partial out_i^{(layer)}} \cdot \frac{\partial out_i^{(layer)}}{\partial net_i^{(layer)}} \tag{6}$$

- if *layer* is the output layer with cross-entropy error function and softmax activation function

$$\partial_i^{(layer)} = \left[\frac{(1-t_i)}{(1-out_i)} - \frac{t_i}{out_i} \right] \cdot \left[out_i^{(layer)} \cdot \left(1 - out_i^{(layer)}\right) \right] \qquad (7)$$

where the term in the first parenthesis is the derivative of cross-entropy error function, and the term in the second parenthesis is the derivative of softmax activation function.

- if *layer* is a hidden layer with sigmoid activation function

$$\partial_i^{(layer)} = \left(\sum_{j=1}^{n} \partial_j^{(layer+1)} \cdot w_{ij} \right) \cdot \left[out_i^{(layer)} \cdot \left(1 - out_i^{(layer)}\right) \right] \qquad (8)$$

where the second parenthesis is the derivative of sigmoid activation function.

2. Compute the variations of the weights:

$$\Delta w_{ij}^{(layer)} = -\eta \cdot \partial_i^{(layer)} \cdot out_j^{(layer-1)} + \beta \cdot \Delta w_{ij}^{(layer)} \qquad \forall j \qquad (9)$$

3. Update the weights:

$$w_{ij} = w_{ij} + \Delta w_{ij} \qquad \forall j \qquad (10)$$

The previous equations for the back-propagation algorithm are summarized in Figure 1, where $e_i^{(layer)} = \dfrac{\partial E_i^{(layer)}}{\partial out_i^{(layer)}}$ is the back-propagated error, associated with node *i*.

$$e_i^{(NLAYERS)} = \left[\frac{1-t_i}{1-out_i^{(NLAYERS)}} - \frac{t_i^{(NLAYERS)}}{out_i^{(NLAYERS)}} \right] \qquad \forall i \qquad (1.1)$$

```
for layer in range(NLAYERS, 1):
```

$$\partial_i^{(layer)} = (1 - out_i^{(layer)}) \cdot out_i^{(layer)} \cdot e_i^{(layer)} \qquad \forall i \qquad (1.2)$$

$$e_j^{(layer-1)} = \sum_{i=1}^{N_{(layer-1)}} w_{ij} \cdot \partial_i^{(layer)} \qquad \forall j \qquad (1.3)$$

$$\Delta w_{ij} = -\eta \cdot \partial_i^{(layer)} \cdot out_j^{(layer-1)} + \beta \cdot \Delta w_{ij} \qquad \forall i \forall j \qquad (1.4)$$

$$w_{ij} = w_{ij} + \Delta w_{ij} \qquad \forall i \forall j \qquad (1.5)$$

Figure 1. Implementation of the back-propagation training algorithm for a MLP with cross entropy error function, softmax output layer units, and sigmoid activation units in the hidden layers.

4. **Speed-up using single and multi-core CPUs**

The first step toward an efficient straightforward implementation of the feed-forward and back-propagation procedures is to use the efficient vector-by-matrix functions.
More efficient implementations are, however, possible after a closer examination of the MLP training procedure. In the batch version of the back-propagation algorithm, the weight variations Δw_{ij} are summed over all the training frames, and the weights are updated once for each epoch. This approach offers the maximum potential of parallel computation, but it has the drawbacks of slow convergence because both the network and the utterance alignments are performed once for each epoch. On the other hand, stochastic gradient back-propagation in which weight updating occurs after a single frame has been processed, provides better accuracy, because it reduces the risk of getting stuck in a local minimum of the error function, and has also a faster learning rate, but does not allow parallel computation to be massively exploited.
An intermediate approach, in which the weights are updated after processing a block of B patterns, is a trade-off solution that leads to accurate model estimation and allows parallel computation. In Bilmes et al. (1997) high performance matrix multiply routines were introduced that, based on the work of Anguita et al. (1994), have been used to improve the training speed on a speech recognition data set.

The training algorithm can be further enhanced by using the so called focused-attention back-propagation (FABP) learning strategy (Hoskins, 1989). FABP focuses attention on the patterns that are most difficult to learn. In FABP, the feed-forward step is performed, as usual, for all the patterns to compute the errors between the net outputs and the related targets. Back-propagation, however, is performed only for those patterns having a Mean Square Error (MSE) greater than a given threshold. This strategy not only speeds-up the training, but also improves the quality of the trained model reducing its dependence on the a priori probability of the classes in the training set.
In FABP several training patterns fed to the network are skipped, i.e. not used for back-propagation (almost 80% in the last iterations). It is, thus, opportune that the feed-forward block size is greater than the one used in the back-propagation step. We will refer to the number of patterns used in the feed-forward step as *bunch size*, FBS, and to the *block size* used in the back-propagation as BBS.
However, the value of FBS must be bounded because the network weights are updated as soon as BBS patterns are accepted by the FABP strategy. Since the network weights have been updated, all the remaining patterns in the feed-forward bunch, accepted by the FABP strategy but exceeding the BBS size, must be submitted again in the next feed-forward bunch, introducing memory and computation overhead.
Taking care of these considerations the bunch feed-forward and the block back-propagation steps can be rewritten using matrix-by-matrix operations as shown in Figure 2 and Figure 3 respectively.

$$\mathbf{out}^{(0)}_{(BxI)} = \mathbf{input}_{(BxI)} \qquad (2.1)$$

```
for layer in range (1, NLAYERS)
```

$$\mathbf{net}^{(layer)}_{(BxI)} = \mathbf{out}^{(layer-1)}_{(BxJ)} \cdot (\mathbf{W}^{(layer)}_{(IxJ)})^T \qquad (2.2)$$

```
    if layer == NLAYERS
```

$$\mathbf{out}^{(layer)}_{(BxI)} = softmax(\mathbf{net}^{(layer)}_{(BxI)}) \qquad (2.3)$$

```
    else
```

$$\mathbf{out}^{(layer)}_{(BxI)} = sigmoid(\mathbf{net}^{(layer)}_{(BxI)}) \qquad (2.4)$$

Figure 2. Implementation of the feed-forward step of the training algorithm for a MLP with softmax output layer units, and sigmoid activation units in the hidden layers, using matrix-by-matrix operations.

$$\mathrm{e}^{(NLAYERS)}_{(BxI)} = (1_{(BxI)} - t^{(NLAYERS)}_{(BxI)}) \cdot / (1_{(BxI)} - \mathrm{out}^{(NLAYERS)}_{(BxI)}) - (\mathrm{t}^{(NLAYERS)}_{(BxI)} ./ \mathrm{out}^{(NLAYERS)}_{(BxI)}) \qquad (3.1)$$

```
for layer in range(NLAYERS, 1)
```

$$\partial^{(layer)}_{(BxI)} = (1_{(BxI)} - \mathbf{out}^{(layer)}_{(BxI)}) \cdot * \mathbf{out}^{(layer)}_{(BxI)} \cdot * \mathbf{e}^{(layer)}_{(BxI)} \qquad (3.2)$$

$$\mathbf{e}^{(layer-1)}_{(BxJ)} = \partial^{(layer)}_{(BxI)} \cdot \mathbf{W}^{(layer)}_{(IxJ)} \qquad (3.3)$$

$$\Delta\mathbf{W}^{(layer)}_{(JxI)} = -\eta \cdot (\mathbf{out}^{(layer-1)}_{(BxJ)})^T \cdot \partial^{(layer)}_{(BxI)} + \beta \cdot \Delta\mathbf{W}^{(layer)}_{(JxI)} \qquad (3.4)$$

$$\mathbf{W}^{(layer)}_{(JxI)} = \mathbf{W}^{(layer)}_{(JxI)} + \Delta\mathbf{W}^{(layer)}_{(JxI)} \qquad (3.5)$$

Figure 3. Implementation using matrix-by-matrix operations of the back-propagation training algorithm for a MLP with cross entropy error function, softmax output layer units, and sigmoid activation units in the hidden layers.
./ and .* represent the element-by-element division and multiplication respectively.

In these figures, *T* indicates transposition, the superscript is the number of the network layer and the subscript gives the dimensions of the matrices, where *B* is the feed-forward bunch size FBS, *I* and *J* are the number of output nodes of two adjacent layers. Introducing a zero dummy layer, the same index refers both to the layer weights and to its outputs. T

The variables η and β in Figure 3, are the dynamic learning rate adjustment and the momentum factor respectively, which, together with some additional techniques illustrated in Sarkar (1995), allow the convergence properties, the training time and the model accuracy to be improved.

$$\mathbf{tmp1}_{(BxI)} = 1_{(BxI)} - \mathbf{t}_{(BxI)} \quad (4.1)$$

$$\mathbf{tmp2}_{(BxI)} = 1_{(BxI)} - \mathbf{out}_{(BxI)} \quad (4.2)$$

$$\mathbf{tmp1}_{(BxI)} = \mathbf{tmp1}_{(BxI)} \cdot / \mathbf{tmp2}_{(BxI)} \quad (4.3)$$

$$\mathbf{tmp2}_{(BxI)} = \mathbf{t}_{(BxI)} \cdot / \mathbf{out}_{(BxI)} \quad (4.4)$$

$$\mathbf{e}_{(BxI)} = \mathbf{tmp1}_{(BxI)} - \mathbf{tmp2}_{(BxI)} \quad (4.5)$$

Figure 4. Decomposing a complex operation in terms of vector operations.

The Level 3 BLAS the Basic Linear Algebra Subprograms (BLAS) (Blackford et. al. 2002) library functions can be used for the matrix-by-matrix multiplications in (2.2), (3.3) and (3.4). In particular, since our experiments were performed on a machine with Intel CPUs, an optimized implementation of the BLAS library for Intel processors, the Math Kernel Library (Intel MKL, 2009) was compiled, which exploits the MMX and SSE instruction sets to speed-up computation.
Moreover, since Intel provides the Integrated Performance Primitives (Intel IPP, 2009) for other vector operations, it is important to decompose operations such as (2.3), (2.4), (3.1), (3.2) and (3.5) in terms of a sequence of vector operations which can be mapped to fast library functions. As an example, the error computation of (3.1) can be decomposed as shown in Figure 4.

Further speed-up is obtained using the threaded MKL implementation of the `cblas_sgemm` function that splits the execution over the available CPU cores.
Since the relative speed-up that can be achieved using these libraries is proportional to the size of the data, large feed-forward bunch size and back-propagation block sizes lead to better performance. However, as mentioned before, the value of FBS must be bounded to reduce the computation overhead. In the experiments described in Section 6 the value of the block size BBS has been set to 10. This value is as a good tradeoff between training time and model accuracy. As a consequence, the value of the feed-forward bunch size FBS has been set to 32 taking into account the statistics about the average number of patterns that are not used for back-propagation in the training iterations .

5. **Speed-up using GPU and CUDA**

GPUs are graphics-oriented dedicated processors suited to computationally expensive but highly parallelizable tasks such as 3D graphic rendering. The main GPU architecture is characterized by the presence of a high number of floating point core processors (up to of some hundreds) which are able to perform parallel computations and can currently reach peaks of the order of one TFLOPS.
Since most of the computation in ANN training has a high degree of fine grain parallelism, the use of GPUs is particularly suited for this task.
The implementation of parallel computation tasks has been remarkably facilitated since

the introduction by NVIDIA of the high-level programming language CUDA (Compute Unified Device Architecture), which provides a C-like interface to the programmable processors of the GPU and an efficient implementation of the BLAS library (CuBLAS).
In the CUDA environment a programmer sees the GPU as a multi-core processor allowing the concurrent execution of multiple threads which perform the same computations on different data. The computation is organized as a grid of thread blocks where each thread executes a single instruction set called kernel (NVIDIA, 2008).
Examples of the use of the CUDA framework in the field of artificial neural networks are already present in literature. In particular, Cernansky (2009) proposes a CUDA implementation of a training algorithm for recurrent networks, Jang et al. (2008) show a possible use of GPU to speed up the feed-forward step, while Lahabar et al. (2008) present a complete implementation of the training process for an MLP with sigmoid output units, assuming that all the training patterns and the corresponding targets are fixed and stored in the GPU memory.
Although our approach is similar to the latter, however, several problems have been taken into account for an effective implementation of the MLP training task for sequential patterns on a GPU. First of all, the epoch back-propagation approach proposed in Lahabar et al. (2008) is not suited for training a network that has to model phonetic units for speaker independent speech recognition tasks. For these tasks the number of training patterns can easily exceed some millions, thus, the patterns cannot be stored in the GPU memory. Furthermore, since slow convergence problems easily arise when epoch back-propagation is used for such large data sets, it becomes mandatory to use the bunch training approach combined with FABP as described in Section 4. Care, however, is required to estimate the bunch and block sizes to achieve a good trade-off between model accuracy and training time. Moreover, the softmax output layer, necessary in our approach to estimate the posterior probabilities of the phonetic unit states, requires the evaluation of the sum of the elements of a vector. This task is not well suited to the multithreaded paradigm. Thus, rather than adopting a concurrent algorithm aimed at minimizing the time required for the sum of the elements of a single array, which requires synchronization barriers, we used instead a simple sequential algorithm for each vector, but the task is performed in parallel over the BBS output vectors of the bunch.

To exploit the GPU computational power, we map the matrix implementation on either CuBLAS functions or specific kernels. Matrix multiplications are performed by means of the `cublasSgemm`, a fast and hardware-optimized implementation of matrix products function, both in the feed-forward (5.2) and in the back-propagation (6.3, 6.4) steps. Carefully tailored kernels have been implemented for the softmax and sigmoid functions, the MSE evaluation, and the update of the network weights. Recall that kernels are set of instructions executed in parallel by all the GPU stream processors. Thus, as an example, a kernel applies the sigmoid function in parallel to all the units of a layer.
Moreover, due to the architecture of the GPU, since CuBLAS give optimal performance with matrices having sizes that are multiples of power of 2, we enforce zero padding to our matrices which do not meet this condition. In particular, we pad to multiples of 32 and 16 the structures for the feed-forward bunch and back-propagation block

respectively, and to multiples of 32 the weight matrices.
Performance improvement of 10% were obtained using padded matrices rather than non-padded ones, as shown in the last two rows of Table 3 in Section 6.
Although the CuBLAS library includes efficient functions to copy data from the host to the card and copy results back from the card to the host, these transfers of data can easily become the performance bottleneck. Thus, the network weights are kept in the GPU memory, but the input patterns are loaded on the GPU in bunches due to the small size of the GPU memory compared to the dimensions of the training data.

The core steps of the training algorithm can be summarized as follows:

1. Load a bunch FBS of input patterns on the GPU memory.
2. Execute the feed-forward step and evaluate the MSE for each pattern.
3. Transfer both the bunch MSEs and output patterns from the GPU to the main memory. The output patterns are required by the forced alignment performed by the Viterbi algorithm, running in the host machine, to obtain a new, more precise, association of the input patterns to the targets. The MSE of each pattern is simply tested in the host processor to select (according to the focused-attention strategy) the patterns on which back-propagation has to be performed.
4. The pattern vectors in the GPU memory selected for back-propagation in step 3 are appended to the back-propagation matrix structures (of size BBS).
5. Repeat all the previous steps until a sufficient number of patterns (the block size BBS) are selected for back-propagation.
6. Execute the back-propagation procedure for the block of selected patterns and update the weights.
7. Since the weights have been updated, all the remaining patterns in the feed-forward bunch that were accepted by the FABP strategy, but exceed the BBS size, must be re-submitted in the next feed-forward bunch.
8. Repeat from the beginning until all input patterns have been processed.

These steps are iterated until convergence is reached as decided by a stopping criterion based on maximum number of iterations, rate of decrease of the MSE, or recognition performance on a held out development set.

Table 1 shows the CUDA functions associated to the MLP training steps detailed in Figure 2 and Figure 3. Four types of functions were employed:

1. Memory transfer CPU/GPU: the function `cudaMemcpy` effectively performs memory transfers between the host and the card memory and vice-versa. It has mainly been used to transfer input patterns from main memory to the GPU global memory for the feed-forward step.
2. Memory transfer GPU/GPU: The pattern vectors in the GPU memory selected for back-propagation need to be copied in contiguous locations into the back-propagation matrix structures. This operation is performed by means of a CUDA kernel.
3. Matrix-by-matrix product: performed by means of `cublasSgemm`.
4. CUDA kernels: all the other operations have been implemented with CUDA

kernels.

Functions	Time (%)	Type
Memory transfer CPU/GPU (2.1)	**8.7**	`cudaMemcpy`
Feed-Forward	**32.5**	
net computation (2.2)	22.8	`cublasSgemm`
softmax function (2.3)	7.2	`kernel`
sigmoid function (2.4)	2.5	`kernel`
Memory transfer GPU/GPU	**10.0**	`kernel`
Back-propagation	**48.8**	
Cross-entropy (3.1)	1.2	`kernel`
Error derivative (3.2)	2.4	`kernel`
Error propagation (3.3)	20.1	`cublasSgemm`
Weight gradient (3.4)	17.0	`cublasSgemm`
Weight update (3.5)	8.1	`kernel`

Table 1. Timing profile for MLP training functions (in parentheses the related formulae in Figure 2 and 3) using CUDA.

Table 1 also shows the timing profile of the training procedure using the CUDA architecture. It is worth noting that most of the time is spent performing actual computations, while the transfer of inputs and outputs between CPU and GPU contributes to the total time for just 8.7%. The overhead arising from the transfer inside the GPU of the output patterns from the feed-forward to the back-propagation structures is significant. This overhead is mandatory because only a fraction of the patterns have to be processed - the ones selected by the focused attention strategy – and they must be stored in physically contiguous areas of memory to allow faster execution of the back-propagation procedure. Excluding memory transfer times, and looking at the actual computations, most of the time (about 74%) is spent evaluating matrix products by means of the CuBLAS library function `cublasSgemm`, whereas the remaining time is used by the kernels. Kernels account for less than one fourth of the global computation time and at least one third of kernel time is actually spent in the weight update kernel, almost all for memory transfers from the GPU main memory to the core processors memory. For this reason we chose not to push the kernel optimizations any further, though some margins for kernel optimization still exist.

6. **Experiments**

A large set of experiments was performed to test the speed-up achieved by using the proposed approaches. The hardware setup was a HP xw8600 workstation equipped with a

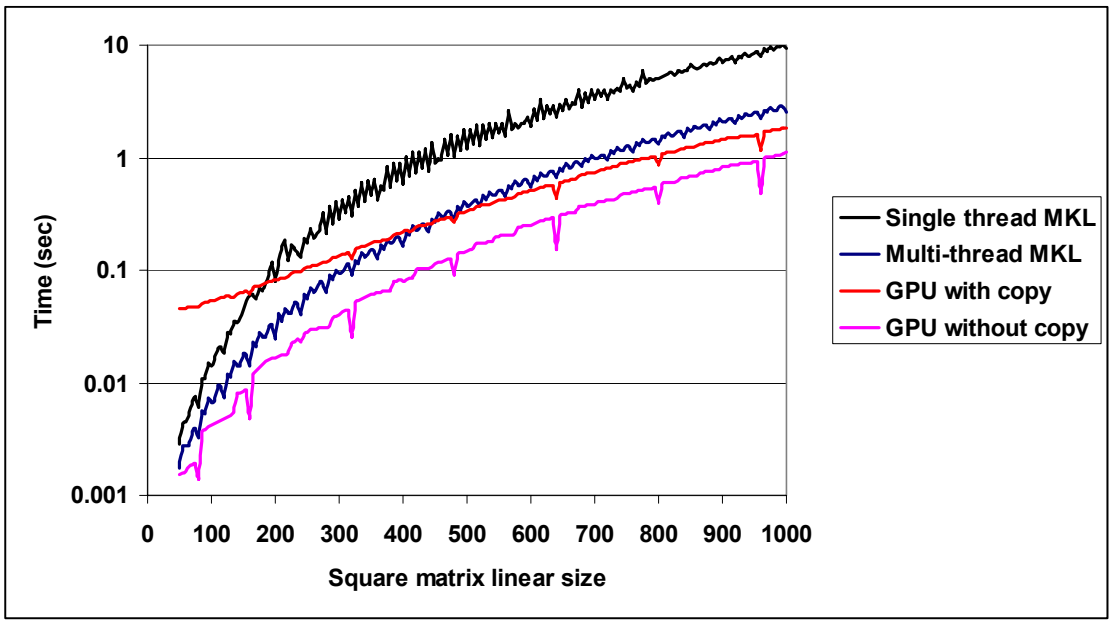

Figure 5. Time required for the product of two square matrices, as a function of their linear size, using different libraries.

quad-core 3.0 GHz CPU, 1600 MHz FSB, 8 GB RAM, NVIDIA GTX280 GPU, and running Linux RedHat RHEL 5.2 EM64T.

Since matrix multiplication is the basic building block for MLP training, in the first set of experiments the performance of the matrix-by-matrix product using the INTEL MKL optimized library was compared with the same operation performed on a NVIDIA GPU. We tested the product of two square matrices with linear dimensions ranging from 50 to 1000 in steps of 5. These products were done by using the well tuned implementation, described in Volkov and Demmel (2009), included in Version 2 of the CuBLAS library.

The results of these experiments are shown in Figure 5 where the execution times are plotted in logarithmic scale. As expected, using a multi-threaded architecture increases the performances of the MKL implementation, more and more for larger matrices. Exploiting the processing power of the GPU allows the CuBLAS implementation to outperform the multi-thread approach using the MKL library for medium-large matrices, even though the overhead for the host/GPU memory transfers cannot be neglected (compare the two plots labelled “with copy” and “without copy”). If we factor out the memory transfer time, CuBLAS performs better than MKL also in the product of small matrices.

This behaviour suggests using large bunch sizes for MLP training for optimizing the memory transfers between the host and the GPU memory to keep the GPU processors busy as much as possible.

It's interesting to observe that the CUDA framework presents performance peaks (of the order of 10%) corresponding to matrix dimensions which are multiples of power of 2, which is the reason for using zero padding.

Training Implementation	Elapsed Time hh:mm	Speed-up versus Standard C	Speed-up versus MKL
Standard C	42:35	-	-
Single thread MKL	11:36	3.7 x	-
Multi-Thread MKL	9:41	4.4 x	1.2 x
GPU and CUDA no-padding	2:29	17.1x	4.7x
GPU and CUDA with padding	2:14	19.1 x	5.2 x

Table 2. Training time, and relative speed-up for the Wall Street Journal 0 corpus.

The second set of experiments was devoted to training English models using the Wall Street Journal corpus (Paul and Baker, 1992), a relatively small database of 7236 files, which represents a popular case study in Automatic Speech Recognition.This corpus has been used as a validation set for the algorithms and for tuning the training procedure.
Table 2 shows the elapsed training time, and the relative speed-up of four implementations: the baseline is a standard C language implementation without the optimized matrix functions offered by the BLAS library, it is compared with single and multi thread programs using the INTEL MKL libraries, and finally with the support of a GPU board and its optimized routines.

It is worth noting that to obtain a fair comparison of the GPU and MKL implementations, the bunch size FBS of the latter has been set to 15 rather than to 32. This has been done to avoid in the MKL implementation the significant overhead of re-processing in the next feed-forward step the patterns accepted by the focused attention mechanism but exceeding the back-propagation block size BBS. This problem is much less relevant in the GPU implementation because bunch of patterns are processed in parallel.
In this case study a speed-up of 3.7 is obtained by enhancing the naïve standard C language implementation to process bunch of patterns using the MKL libraries. The improvement obtained using the GPU is much larger: a factor of 19.1, which is 5.2 times faster than the single thread MKL implementation. Although the multi-thread MKL version does give a 20% improvement compared to the single thread, it fully occupies the 4 cores, which can be used instead for training in parallel different models.

In the third set of experiments, without changing the setup, the models of 6 different languages have been re-trained using the large corpora collected for creating the models used by the Loquendo decoder. The training process has been performed using the same corpora, number of iterations and training parameters, excluding the bunch size. It is worth noting that if the focused attention strategy is disabled, the bunch size FBS does not have any influence on the update of the network weight, it is only used for efficiency purposes. Only the back-propagation block size BBS has a direct impact on final model because it drives the rate of the weight updates. On the contrary, the bunch size affects the final model parameters if the focused attention strategy is enabled. This happens because the feed-forward step is done for the set of patterns belonging to a bunch using

the current network weights. The patterns that are not selected for back-propagation will not be included in the next bunch. One or more of these patterns could be possibly

Languages	Loquendo ASR Models	Single thread MKL implementation			GPU-CUDA implementation			
	WA (%)	WA (%)	ΔERR	Time hh:mm	WA (%)	ΔERR	Time hh:mm	Speed-up
Italian	93.1	93.0	+0.1	143:28	93.0	+0.1	29:26	4.9 x
Spanish	93.2	93.3	-0.1	60:55	93.2	0.0	11:23	5.4 x
French	90.2	90.2	0.0	107:37	90.0	+0.2	16:39	6.5 x
German	89.4	89.7	-0.3	94:00	90.0	-0.6	13:21	7.0 x
English	84.6	84.5	+0.1	147:31	84.1	+0.5	24:30	6.0 x
Brazilian	84.1	84.7	-0.6	50:47	84.2	-0.1	7:33	6.7 x
AVERAGE	89.1	89.2	-0.1		89.1	0.0		6.1 x

Table 3. Training time of the acoustic models of 6 different languages with MKL and GPU-CUDA implementations, and average accuracy for 8 application grammars.

selected, instead, if the bunch size were smaller because it would be classified with a different set of network weights.
Thus, the models resulting from the MKL and GPU system are slightly different, although, as it will be shown, their accuracy is comparable.
In these experiments we computed the elapsed training times to evaluate the obtained speed-up and we checked the recognition accuracy on 8 different recognition application grammars to assess that fast computation does not produce statistically significant recognition performance variations. Tests were performed for utterances collected both from the fixed and the mobile telephone networks.
The grammars include common recognition tasks such as yes-no, connected digits, numbers, spelling, dates, etc. The reference models are the ones released with the Loquendo ASR recognizer. The models were re-trained using the single thread MKL implementation on the same workstation hosting the NVIDIA GPU to obtain a fair comparison of the training times.

Table 3 shows experimental results comparing the standard Loquendo ASR models, models retrained using the single thread MKL implementation, and models retrained exploiting the GPU-CUDA architecture. The percent Word Accuracy (WA) in the Table gives the average results obtained using similar application grammars in different languages. The comparison of the results among languages is not significant because the list of the grammars slightly differs. The table gives also the elapsed training time, in hours and minutes, for the retrained models (MKL and CUDA-GPU implementations), and, in the last column, the relative speed-up of the GPU-CUDA with respect to the MKL implementation.
The results in this table are useful to evaluate the average speed-up in training and Word Accuracy in recognition obtained with the GPU-CUDA implementation in comparison

with a traditional, but MKL optimized training. The obtained average speed-up is more than 6 times keeping substantially constant the accuracy averaged on all the languages. The difference in speed-up for the various languages is due to the different amount of

Experiment	GTX280		GTX295(1)		GTX295(2)	
	Time	Incr (%)	Time	Incr (%)	Time	Incr (%)
1	2:14	-	idle		idle	
2	idle		2:20	+4.5	idle	
3	2:14	0	2:20	+4.5	idle	
4	idle		2:20	+4.5	2:20	+4.5
5	2:15	+0.7	2:22	+6.0	2:21	+5.2

Table 4. Training time, in hours and minutes, for single and parallel training sessions running on different GPUs.

speech data used in the training and to the number of phonetic units that are defined for each language, and as a consequence, to size of the corresponding MLP networks.
Finally, an additional NVIDIA GTX295 GPU board has been added to the hardware configuration to test the possibility of training more networks in parallel. This board has two GPUs, but a lower clock both for its memory and core processors. In this configuration the workstation has 3 GPUs of comparable computational power.
A set of training sessions have been done using the Wall Street Journal corpus to verify whether the simultaneous run of more than one training session slows down the GPU-CUDA implementation, possibly generating congestion problems on the PCI-Express bus. The training sessions have not been done on larger corpora due to the limitations of the hardware configuration for these experiments, offering 8 GB of main memory only. A large increase of the training sets would impact on the elapsed times for the overhead introduced by the use of virtual memory. Being sure that the input data can be stored in main memory eliminates this factor, focusing the experiments on the performance of the GPU boards, and on their interactions with the main memory.
Table 4 shows the elapsed times, in hours and minutes, for training the 7236 sentences of the corpus. Each row in the Table represents experiments for a single or parallel training sessions running on different GPUs, the columns show the training time for the active GPUs.
Rows 1 and 2 compare the performance of the two GPU boards on the single model training experiments. It is not surprising that the training executed on one GPU of the GTX295 board is slightly more time consuming, due to its slower board clock.
In rows 3 and 4 we report the results of running two training sessions on the same corpus, using one GPU of the two boards or the two GPUs of the GTX295 board respectively. In both cases the training times do not increase compared to the single training session. Thus, the two parallel training sessions do not interfere, and no bandwidth problems arise for the PCI-Express 16x Gen2 bus.
As shown in the last row, this configuration of the workstation is able to carry out three parallel sessions with a very small increase of the training time, of the order of 1%. We can conclude that the GPUs and the PCI-Express 16x Gen2 bus are not a bottleneck for the fast GPU-CUDA training implementation.

7. **Conclusions**

We have studied and implemented a complete ANN training procedure for sequential patterns exploiting the computational power of inexpensive GPU boards. We have tested this approach for training acoustic models for large vocabulary speech recognition tasks, showing a 6 times reduction of the time required to train real-world large size networks with respect to an already optimized implementation using the INTEL MKL libraries.
Thanks to these optimizations and to the support of the GPU, the training time for language having a huge set of training sentences (about one million for Italian) can be reduced from approximately a month to 5 days.
The obtained speed-up not only improves the efficiency of the generation of acoustic models, but also makes easier the research activity related to acoustic modeling such as testing different definitions of the acoustic units, experimenting with different neural networks structures and topologies. Reducing the training time also allows training larger models with huge training corpora.

References

Albesano D., Gemello R., Mana F., 1997. Hybrid HMM-NN Modeling of Stationary-Transitional Units for Continuous Speech Recognition. In: Proc. Internat. Conference on Neural Information Processing, pp. 1112–1115.

Anguita D., Parodi G., Zunino R., 1994. An efficient implementation of BP on RISC-based workstations. Neurocomputing, Vol. 6, pp.57-65.

Bilmes, J., Asanovic, K., Chin C., Demmel, J., 1997. Using PHiPAC to speed error back-propagation learning. In: Proc. Internat. Conference on Acoustics, Speech and Signal Processing, ICASSP-97, pp .4153-4156.

Blackford L. S. et al., 2002. An Updated Set of Basic Linear Algebra Subprograms (BLAS), ACM Transactions on Mathematical Software, Vol. 28, n.2 , pp. 135-151.

Bourlard H., Morgan N., 1993. Connectionist Speech Recognition: A Hybrid Approach. Kluwer Academic Publishers.

Cardinal P., Dumouchel P., Boulianne G., 2009. Using Parallel Architectures in Speech Recognition. In Proc. INTERSPEECH 2009, pp. 3039–3042.

Cernansky M., 2009. Training Recurrent Neural Network Using Multistream Extended Kalman Filter on Multicore Processor and Cuda Enabled Graphic Processor Unit. In: (C. Alippi et al. (Eds) Lecture Notes in Computer Science. Artificial Neural Networks – LNCS 5768, Part 1, ICANN 2008, Limassol, Cyprus, pp. 381–390.

Chan A., Sherwani J., Mosur R., Rudnicky A., 2004. Four-Layer Categorization Scheme of Fast GMM Computation Techniques in Large Vocabulary Continuous Speech Recognition Systems. In: Proc. INTERSPEECH 2004, pp. 689–692.

Dixon P.R., Oonishi, T., Furui, S., 2009. Harnessing graphics processors for the fast computation.of acoustic likelihoods in speech recognition. Computer Speech and Language 23, pp. 510–526.

Fissore L., Laface P., Ravera F., 1995. Acoustic-Phonetic Modeling for Flexible Vocabulary Speech Recognition. In: Proc. EUROSPEECH 95, pp. 799-802.

Hertz J., Krogh A., Palmer R.G., 1991. Introduction to the Theory of Neural Computation. Addison Wesley.

Hoskins, J.C., 1989. Speeding Up Artificial Neural Networks in the "Real" World. In: Proc. Internat. Joint Conference on Neural Networks, p. 626.

Intel MKL: Math Kernel Library, 2009. http://software.intel.com/en-us/intel-mkl

Intel IPP: Integrated Performance Primitives, 2009. http://software.intel.com/en-us/intel-ipp

Jang, H., Park, A. Jung, K., 2008. Neural Network Implementation Using CUDA and OpenMP. In: Proc. Digital Image Computing: Techniques and Applications, DICTA '08, pp.155-161.

Lahabar S., Agarwal P., Narayanan P. J., 2008. High Performance Pattern Recognition on GPU. In: Proc. National Conference on Computer Vision Pattern Recognition, Image Processing and Graphics, NCVPRIPG'08, pp. 154-159.

Loquendo ASR: http://www.loquendo.com/en/technology/asr.htm

NVIDIA: NVIDIA CUDA Compute Unified Device Architecture Programming Guide, 2008. http://developer.download.nvidia.com/compute/cuda/2_0/docs/ CUBLAS_Library_2.0.pdf

Oh, K.S., Jung K., 2004, GPU implementation of neural networks. Pattern Recognition, Vol. 37, pp.1311-1314.

Paul D.B., Baker J.M., 1992. The Design of the Wall Street Journal-based CSR Corpus. In: Proc. Intern. Conference on Speech and Language Processing, ICSLP-1992, pp. 899-902.

Rabiner L., Juang B.H., 1993. Fundamentals of Speech Recognition. Prentice-Hall.

Rumelhart, D.E., Hinton, G.E., Williams, R.J., 1986. Learning Internal Representations by Error Propagations. Parallel Distributed Processing: Explorations in the Microstructure of Cognition. Vol. 1. Foundations, pp. 318-362, MIT Press.

Sarkar, D., 1995. Methods to Speedup Error Back Propagation Learning Algorithm. ACM Computing Surveys, Vol. 27, n. 4, pp. 519-542.

Volkov V., Demmel J.W., Benchmarking GPUs to tune dense linear algebra. In: Proc. of the 2008 ACM/IEEE Conference on Supercomputing, 2008, pp. 1–11.